UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Open source tools for container vulnerability analysis

Master's Thesis

Master's Degree in Informatics Engineering

AUTHOR: Pardo Minaya, Sergio David

Tutor: Andrés Martínez, David de

External cotutor: Baiardi, Fabrizio

ACADEMIC YEAR: 2023/2024

Abstract

Containerization technologies have been an important element in the spread of cloud computing adoption, since they have enabled the deployment of modern applications in a more efficient way. They offer several advantages: container virtualization mechanisms are much less resource intensive compared to traditional virtualization technologies, containers have great portability, since application code and libraries are packaged together in a standardized way, they allow developers to build applications based on microservices much more easily, and they still maintain great security and isolation of the applications running inside.

Even though containers can offer a good level of security and isolation, the flexibility of containerization technologies means that their optimal configuration is a non-trivial task, which, together with bugs present in any of its layers, can end up creating vulnerabilities that can be potentially exploited by malicious actors. Security breaches are a central issue in the current technological landscape and are gaining increasing attention due to the amount of them that happen every year and their nefarious consequences for confidentiality, integrity and availability.

Container vulnerabilities can be broadly classified in container application vulnerabilities, container configuration vulnerabilities, container image vulnerabilities, container engine vulnerabilities and host vulnerabilities. In this project, we aim to study each of these types of vulnerabilities, search the open source tools available to tackle each of them and compare a set of such tools to determine which ones are more adequate.

Keywords: container; image; Docker; virtualization; application; security; integrity; confidentiality; availability; vulnerability; analysis; tool; free software; open source software; detection.

Resum

Les tecnologies de contenidors han sigut un element important en la difusió de l'adopció de la computació al núvol, donat que han permés el desplegament d'aplicacions modernes d'una manera molt més eficient. Oferixen diversos avantatges: els mecanismes de virtualització de contenidors utilitzen considerablement menys recursos comparat amb les tecnologies de virtualització tradicionals, els contenidors tenen una gran portabilitat, ja que el codi de l'aplicació i les biblioteques són empaquetats junts d'una manera estandarditzada, permeten als desenvolupadors crear aplicacions basades en microserveis molt més fàcilment, i tot això mantenint un bon nivell de seguretat i aïllament de les aplicacions que s'hi executen.

Tot i el bon nivell de seguretat i aïllament que els contenidors poden oferir, la flexibilitat de les tecnologies de contenidors implica que la seua configuració òptima no és una tasca gens trivial, cosa que, junt amb els errors presents en qualsevol de les seues capes, pot acabar creant vulnerabilitats que poden ser potencialment explotades per actors malintencionats. Les fallades de seguretat són un assumpte central en el panorama tecnològic actual i estan guanyant una creixent atenció degut a la quantitat d'elles que ocorren cada any i a les conseqüències nefastes que tenen per a la confidencialitat, la integritat i la disponibilitat.

Les vulnerabilitats dels contenidors poden ser classificades de manera general en vulnerabilitats de les aplicacions dels contenidors, vulnerabilitats de la configuració dels contenidors, vulnerabilitats de les imatges dels contenidors, vulnerabilitats del motor de contenidors i vulnerabilitats del sistema amfitrió. En aquest projecte, es pretén estudiar cadascun d'aquests tipus de vulnerabilitats, cercar les ferramentes de codi obert disponibles per a abordar cadascuna d'elles i comparar un conjunt de tals ferramentes per a determinar quines són les més adequades.

Paraules clau: contenidor; imatge; Docker; virtualització; aplicació; seguretat; integritat; confidencialitat; disponibilitat; vulnerabilitat; anàlisi; ferramenta; programari lliure; codi obert; detecció.

Resumen

Las tecnologías de contenedores han sido un elemento importante en la difusión de la adopción de la computación en la nube, dado que han permitido el despliegue de aplicaciones modernas de una manera mucho más eficiente. Ofrecen varias ventajas: los mecanismos de virtualización de contenedores utiliza considerablemente menos recursos comparado con las tecnologías de virtualización tradicionales, los contenedores tienen una gran portabilidad, ya que el código de la aplicación y las bibliotecas son empaquetados juntos de una manera estandarizada, permiten a los desarrolladores crear aplicaciones basadas en microservicios mucho más fácilmente, y todo eso manteniendo un buen nivel de seguridad y aislamiento de las aplicaciones que se ejecutan.

A pesar del buen nivel de seguridad y aislamiento que los contenedores pueden ofrecer, la flexibilidad de las tecnologías de contenedores implica que su configuración óptima no es una tarea nada trivial, cosa que, junto con los errores presentes en cualquiera de sus capas, puede acabar creando vulnerabilidades que pueden ser potencialmente explotadas por actores malintencionados. Los fallos de seguridad son un asunto central en el panorama tecnológico actual y están ganando una creciente atención debido a la cantidad de ellos que ocurren cada año y a las consecuencias nefastas que tienen para la confidencialidad, la integridad y la disponibilidad.

Las vulnerabilidades de los contenedores pueden ser clasificadas de manera general en vulnerabilidades de las aplicaciones de los contenedores, vulnerabilidades de la configuración de los contenedores, vulnerabilidades de las imágenes de los contenedores, vulnerabilidades del motor de contenedores y vulnerabilidades del sistema anfitrión. En este proyecto, se pretende estudiar cada uno de estos tipos de vulnerabilidades, buscar las herramientas de código abierto disponibles para abordar cada una de ellas y comparar un conjunto de tales herramientas para determinar cuáles son las más adecuadas.

Palabras clave: contenedor; imagen; Docker; virtualización; aplicación; seguridad; integridad; confidencialidad; disponibilidad; vulnerabilidad; análisis; herramienta; software libre; código abierto; detección.

Table of Contents

List of Figures

List of Tables

1. Introduction

In this chapter we describe the motivation behind this project and the goals we aim to achieve with it. Furthermore, we also describe the methodology followed for the elaboration of the theoretical and practical sections and detail the structure of the project.

1.1. Motivation

The security of computer systems has been an essential issue since the beginning of our computer-governed society. At their conception, computers were not designed specifically to be secure, but to work correctly. Because of this, when their use started spreading, it was obvious that there was a serious problem that needed to be addressed. Historically, cybersecurity has tried to follow the footsteps of technological innovation, since new technologies often introduce unintended side effects that only time after are noticed, understood and solved.

The prominent opinion in the current cybersecurity space is that *security by obscurity*, that is, when the security of a system is based in hiding and obfuscating its source code, is not a good approach in the long term. Instead, having the source code publicly available and checked by as many people as possible is the most effective way of finding flaws that would have gone unnoticed otherwise. The cybersecurity community appreciates the usage of open source programs that can be audited more thoroughly and transparently, which in turn allows to better understand and trust the results obtained. For this reason, we consider that a good way to contribute to the adoption of open source tools is by testing their effectiveness in various scenarios.

In the world of virtualization, containers are by no means a new technology, but their usage has gone mainstream quite recently following a trend of designing software based on a microservice architecture. Therefore, this project is motivated because it is not always easy to understand the intricacies of containerization technologies and all the ways in which weaknesses can appear, so it can be helpful to get to know their functioning in depth and determine which are the best practices and tools that can provide the best coverage against security incidents.

1.2. Goals

The goal of this project is to understand in a more comprehensive way all the layers and components that make up a modern implementation of a containerization engine and the ways and tools that can be used in order to provide a good security coverage.

1.3. Methodology

On the one hand, to carry out the theoretical research needed for this project, we have obtained information from a variety of sources such as books, scientific papers, articles and project repositories and documentation sites.

On the other hand, to carry out the practical tests for the evaluation of the container security tools, we have employed a set of vulnerability scanners and a container engine like Docker to manage the life cycle of container images.

1.4. Structure

This project is divided into seven main sections that we will detail below.

The first part offers an introduction to the project and explains which are the main goals and motivations behind it.

The second part constitutes a walk through the technological context that has an impact on the project, which is made up of three main areas: virtualization technologies, computer security and free and open source software.

The third part delves into the architecture of a modern containerization implementation and explains various aspects of its internal workings and functions.

The fourth section describes the different types of vulnerabilities involved in the security of containers and gives examples of each of them.

The fifth section offers an exposition of some of the most relevant open source tools for the security of containers and explains their capabilities.

The sixth section describes the preparation and results of the tests carried out to compare the efficacy of a set of image vulnerability scanners.

Finally, the seventh part offers a series of conclusions from the work done throughout the project.

2. State of the art

This project lies at the intersection among different areas in computer science, namely how to improve the *security* of *containers* by means of *open source* tools for *vulnerability* analysis. In order to better understand how much progress we have achieved in each area, we will elaborate on the evolution of each one in the following subsections.

2.1. Virtualization technologies

Virtualization is a technology that enables the creation of multiple simulated environments or resources from a single physical hardware system (1). The need for the virtualization of computing resources has been around for nearly as long as computers themselves. In the 1960s, computers were mainly made up of external equipment used for input and output and internal arithmetic circuits, which had a huge difference in speed (2). This made computers extremely inefficient, and so a paper titled *Time sharing in large fast computers*, written by C. Strachey in 1959, encouraged the adoption of a time sharing paradigm. It offered the main advantage of allowing many programs to run at the same time, which in turn helped reduce the computer's idle time and gave it the ability to overlap computation and I/O operations. However, in order to achieve this new resource management, it was necessary to introduce a layer between the proper hardware and programs, which was called *Director*, one of the first ideas of the necessity of the modern concept of operating system.

Time sharing created the base on which the first attempts of virtualization started to appear. The initial objective of managing the computer resources among different processes led to the appearance of more complex ideas like memory protection and isolation to prevent programs from interfering with each other. Soon, these concepts got developed even further, so that it could be possible to isolate resources of the host machine in such a way as to create the illusion that those resources conformed a real physical machine (3).

The first attempts at implementing resource virtualization appeared in the mid 1960s and early 1970s with some IBM machines like M44/44X or System/360 that already showed one of the key advantages of virtualization technology, that is, the possibility of using a single physical machine to test and execute programs in different environments (including the operating system).

Also during the 1960s and 1970s, the inclusion of the *chroot* Unix system call that allows for file system *namespace* isolation is said to have set the origin of containers, at that time called *capabilities*. However, these initial capabilities lacked many kinds of isolation techniques that are present in current technologies due in part to the complexity of implementing them with the hardware of that time.

During this time, companies had mainframes to which users connected by means of light terminals, thus making it useful to isolate user tasks among them. However, during the 1980s personal computers started to spread, which meant that terminals, and thus mainframes, were not as commonly used anymore (4). This, together with the difficulty of hardware to support efficient implementations, their complexity and performance issues made virtualization to be left aside.

Modern virtualization technologies started appearing in the late 1990s thanks to a paradigm shift in the way that resources were consumed: the advent of the Internet. When companies started putting their websites online, they usually hosted them on their own servers. However, as smaller companies and individuals started creating their own websites, it was very prohibitive to build and maintain their own physical server. Soon, a set of companies emerged with the purpose of letting others rent a subset of their server resources in order to allow them to publish their content online. This brought back the necessity to isolate the resources assigned to each customer and reengaged the development of virtualization technologies such as VMware, Xen, KVM or Hyper-V. Still, there were some issues like the lack of full support by the x86 architecture, which made it necessary to add extensions, and the struggle to find the right balance between security and performance.

Also during the 1990s, but more extensively during the 2000s, key technologies for container implementations started to appear. More specifically, process isolation and resource control, which enabled the isolation of file systems, processes, the network stack and users was introduced in the Linux kernel via *namespaces* and *cgroups* (5) (6). Another set of key features was added to the Linux kernel, such as a framework called Linux Security Module that allowed to load security extensions to the kernel as modules. This framework enabled other security extensions such as AppArmor and SELinux, both conceived at the beginning of the 2000s, to insert access control checks. Also to be noted is the inclusion of a set of patches to the Linux kernel called *seccomp*, which allows for further process restrictions.

All these technologies in the Linux kernel propelled the development of modern container solutions during the 2010s such as Docker, LXC or Podman by allowing the usage of kernel features. Also during this time, an entire ecosystem of applications and services offered through the Internet started to mature, which introduced new software development paradigms such as microservice architectures that benefited greatly from container technologies. However, as these applications grew more complex and sophisticated, surged the need for an orchestration technology, which gave birth to tools such as Docker Swarm, Kubernetes or LXD.

Although the evolution of virtualization and containerization technologies has been intertwined for a long time, it is important to consider the key differences of their modern implementations in order to understand their implications on two key areas: performance and security.

Both virtualization and containerization create an abstraction layer over computer physical resources in order to divide them into isolated bundles. The main technical difference between both concepts lays in the way such isolation is carried out.

On the one hand, virtualization relies on a hypervisor, a software layer that coordinates the division and assignation of resources among virtual machines. Hypervisors can be of type 1 if they interact directly with the physical resources without the presence of a host operating system, or of type 2 if, instead, they lay on top of a host operating system. Virtual machines are virtual environments that simulate a physical computer in software form and usually require the installation of a guest operating system on them in order to be functional, which introduces duplicities in the software stack that have a huge penalty on performance (7).

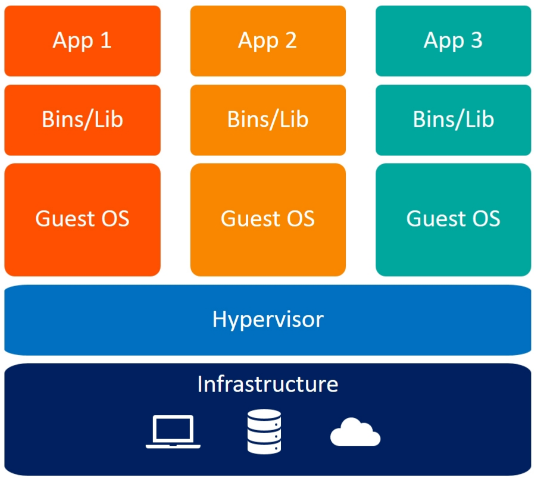

Figure 1: Virtualization software stack[1]

On the other hand, containerization appears as as means of smoothing that performance hit of virtualization by reducing the size of the software stack. Thus, instead of

1 https://raw.githubusercontent.com/dazzyddos/dazzyddos.github.io/master/Images/DockerBuildSec/vmvsdocker.jpg

virtualizing a whole operating system, the virtualized environment, now called container, makes use of the underlying host operating system kernel and creates an isolated environment on top of it, which comprises libraries and applications. This way, containers can improve performance at the cost of security, since they usually offer less isolation capabilities than virtual machines.

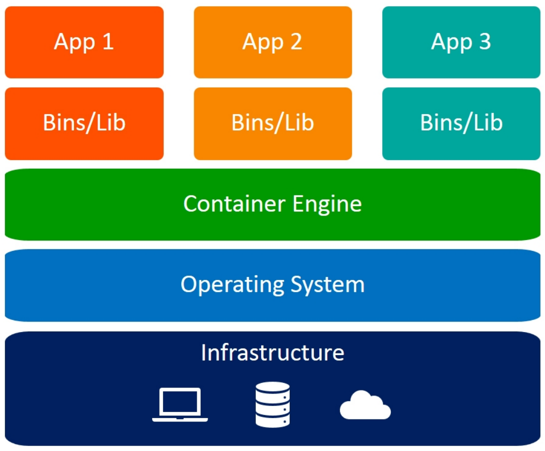

Figure 2: Containerization software stack[2]

2.2. Computer security

Computer security refers to the practice of ensuring confidentiality, integrity and availability of systems and data (8). The history of computer security is also a long one, and dates back to the origins of computers. The first instances of hacking did not even happen on computers, but on phones back in the 1960s in order to make free long distance calls, which shows that tinkering and pushing the design limitations of technology might be an innate instinct of humanity. During this time, the main focus in computer security was physical access, since it was mostly the only way in which computers of that time could be compromised (9).

In the 1970s ARPANET appeared, which was used as a testing environment for the development of what would later become the Internet. At this time, some researchers started developing one of the first computer virus and antivirus, Creeper and Reaper. Later on in the 1980s, the Internet started being deployed and brought with it a greater interconnection among the academic and governmental computers of the time. This was the perfect environment for the first cybersecurity incidents to take place, like the leak of military US documents that ended up in the hands of Russia's KGB.

2 https://raw.githubusercontent.com/dazzyddos/dazzyddos.github.io/master/Images/DockerBuildSec/vmvsdocker.jpg

The 1990s and 2000s saw the widespread adoption of the Internet in the consumer space, which opened a unique opportunity for hackers to start sharing malware and carrying out large scale cyberattacks that had nefarious consequences not only for individuals but also for companies. This led to the appearance of the first modern commercial antivirus software and forced the entire Information Technology (IT) industry to take computer security seriously.

During 2010s, the first state sponsored attacks started being carried out, which further increased the damage of security incidents on governments and enterprises. Additionally, the widespread adoption of smartphones, the rise of Internet of Things (IoT) devices or the generalized use of cloud computing made modern society even more dependent on online services and applications.

Before we continue, it is important to keep in mind the meaning of some concepts:

- A *weakness* is a condition in a software or hardware component that, under certain circumstances, could contribute to the introduction of vulnerabilities (10).
- A *vulnerability* is an exploitable instance of one or more weaknesses that enables an attacker to cause a negative impact to the confidentiality, integrity or availability of an application (11).
- An *exploit* is a piece of code designed to take advantage of one or more vulnerabilities in order for an attacker to gain access or control of a system (12).

In order to improve the distribution and collaboration among organizations and researchers of information about security incidents, a series of public databases were created. One of them is the Common Weakness Enumeration (CWE) (10), a community developed list of common software vulnerability types with descriptions and guidance on mitigations. A complement to the CWE is the Common Vulnerabilities and Exposures (CVE) (11), a standardized system for identifying and tracking publicly known vulnerabilities of specific products or systems. The Known Exploited Vulnerabilities (KEV) (13) is a subset of the CVE database created in 2021 by the US Cybersecurity & Infrastructure Security Agency (CISA) agency that focuses on vulnerabilities in software, hardware, applications or systems that are actively being exploited by threat actors. This list aims to help organizations prioritize their vulnerability management procedures (14). Also important to mention is the Common Vulnerability Scoring System (CVSS) (15), a standardized framework to assess the severity and impact of cybersecurity vulnerabilities. The goal of this scoring system is to help the security community by enabling the prioritization of vulnerability responses, making vulnerability assessments objective and consistent and allowing an effective communication.

If we dive deeper into weaknesses, they can be categorized in many different ways. The CWE offers a *research view* (16) that is useful for understanding their classification from a high level perspective. According to it, the top level categories are:

- *Improper Access Control*: Lack of or incorrect access restriction to a resource for an unauthorized actor.
- *Improper Interaction Between Multiple Correctly-Behaving Entities*: Two entities that behave correctly when running independently behave incorrectly when integrated into a larger system.
- *Improper Control of a Resource Through its Lifetime*: The control over a resource throughout its lifetime of creation, use and release is incorrectly maintained.
- *Incorrect Calculation*: A calculation generates incorrect or unintended results that are later used in security sensitive decisions or resource management.
- *Insufficient Control Flow Management*: The control flow of the code is not sufficiently managed during execution, creating conditions in which it can be modified in unexpected ways.
- *Protection Mechanism Failure*: Protection mechanisms that provide sufficient defenses against attacks are not used or are used incorrectly
- *Incorrect Comparison*: Two entities are compared in a security relevant context, but the comparison is incorrect, which may produce a weakness.
- *Improper Check or Handling of Exceptional Conditions*: Exceptional conditions that rarely occur during normal operation are not properly handled.
- *Improper Neutralization*: It is not correctly ensured that structured data is well formed and that security properties are met before being read or sent.
- *Improper Adherence to Coding Standards*: Development coding rules are not followed, which can lead to weaknesses or increase the severity of vulnerabilities.

Those categories are further subdivided in order to describe weaknesses in more detail. For instance, some of the most common and impactful software weaknesses described in the *2023 CWE Top 25* (17) are: *out-of-bounds write*, *out-of-bounds read*, *improper input validation*, *use after free*, *NULL pointer dereference*, *missing authorization*, *use of hard-coded credentials*, etc.

In the scope of container technologies, there are different types of vulnerabilities. If we go from top to bottom in the software stack, first we can find *application vulnerabilities*, which are vulnerabilities associated with flaws present in binary and library files, as well as badly configured settings of an application. Next, we find *configuration vulnerabilities*, which are associated with flaws in the container settings related to permissions, storage and networking that could allow unexpected behaviors to occur. Then, we have *image vulnerabilities*, which are due to flaws in container images. Moreover, *container engine vulnerabilities* are vulnerabilities associated with flaws present in container virtualization engines such as Docker or Podman. For instance, isolation vulnerabilities could allow attackers to get out of the virtualized environment. Finally, we have *host vulnerabilities*, which are associated with flaws present in the host machine where containers run. These vulnerabilities include improper operating system settings or vulnerable system software.

In order to tackle the complex task of managing the correct detection and patching of such a wide array of vulnerabilities, specific tools called *vulnerability scanners* can be of great help. Vulnerability scanners are automated tools that check computer networks, systems and applications for signs of security weaknesses that could be exploited by hackers (18). Vulnerability scanners generally run a series of tests that are performed and tuned based on the scope of the analysis carried out. The main types of vulnerability scanners in the scope of containerization are *application vulnerability scanners*, *configuration vulnerability scanners*, *image vulnerability scanners*, *network vulnerability scanners* and *host vulnerability scanners* (19) (20) (21).

Application vulnerability scanners scan applications and websites in search of misconfigurations and vulnerabilities with a focus on the view that users have of the system, which can be useful to understand how much a system is exposed depending on the user role (and, therefore, authentication privileges). Furthermore, they can be divided into two types: static and dynamic. Static application vulnerability scanners are executed before the application is run, while dynamic application vulnerability scanners stay active and monitor the application for the duration of its execution.

Configuration vulnerability scanners are tools that analyze the files used to configure the deployment of containers, like Dockerfiles, and alert of any kind of settings that could be problematic or create risks for the security of containers, such as volume and network settings or the presence of secrets stored in an unsafe way.

Image vulnerability scanners retrieve an image from the repository, decompose it into its different layers and examines each layer in search of vulnerabilities, be it in the packages or the libraries present in it. In order to detect vulnerabilities, these tools query some databases like the CVE and check for indicators that allow to identify them.

Network vulnerability scanners can be deployed on physical or virtual machines in order to scan networks by sending probes looking for open ports and services and testing each service afterwards in search of misconfigurations, weaknesses or vulnerabilities. They can be further divided into two types: internal and external. Internal network vulnerability scanners focus on analyzing the private network of the organization, whereas external network vulnerability scanners analyze the network from the outside.

Host vulnerability scanners are deployed as agents on each of the machines of interest in order to scan for local vulnerabilities in the system, with a focus on the operating system, libraries and programs running on the host. The information gathered is then usually sent to a central server for analysis. They can also be divided into static and dynamic scanners, depending on whether they run just once or supervise constantly the state of the host.

2.3. Free and open source software

The birth and evolution of the concepts of free and open source software have had a huge influence in the way software is developed and distributed. Up until the end of the 1970s, there was a widespread culture among universities and computer science communities of unrestricted sharing of software alongside its source code, mainly to facilitate the exchange of ideas and ways to improve or reuse parts of the program (22).

The early years after the birth of the Unix operating system as a research project at AT&T's Bell Laboratories in 1969 consisted in a collaborative development and sharing among computer scientists, effectively acting as free and open source software at a time when such concept did not exist yet. The following years, the development of Unix gained traction thanks in part to the contribution of ideas by programmers from all over the world and the licensing for educational institutions.

This climate of openness started to change in the 1980s when, in 1984, AT&T initiated the commercialization of Unix, which meant that it was no longer a freely shared product. Therefore, universities did no longer have permission to share the source code in educational contexts and it became much harder for developers to share and collaborate on code. Another consequence of the software commercialization trend in the 1980s was that software stopped being distributed with binary and source code files together as it had been the norm until then. Now, companies such as IBM were spreading the practice of only distributing binary files, which prevented programmers from modifying or studying the functionality and effectively made software closed source.

It is important to understand, though, that the underlying problem was not, for instance, that AT&T was making money with Unix licenses, but the fact that selling the software had become the end commercial goal. This implied a shift in perspective from carrying forward computer science knowledge to merely generating a revenue stream from selling software.

As a response to this movement of commercialization and closed source software, something that he had experienced personally at the Massachusetts Institute of Technology (MIT) lab where he worked, Richard Stallman announced in 1983 the creation of the GNU’s Not Unix (GNU) project, which aimed to create a complete Unix compatible operating system and distribute it freely. During the first years, the GNU project had a slow pace of development due to the skepticism of its feasibility. However, by the 1990s there were already donations from several companies and individuals, and work on many programs of the operating system were underway.
In 1985, Stallman created the Free Software Foundation (23), which provided institutional grounding and enabled to include GNU in a bigger initiative to promote free software. Around the same time, he also published the *GNU Manifesto (24),* a document stating his vision for the GNU project, and the *Free Software Definition* (25), where he defines what is free software and which are the fundamental freedoms that should be inherent to it . Therefore, according to Stallman a piece of software is free if users have the following four freedoms (26):

- Freedom to run the program for any purpose.
- Freedom to study how the program works and adapt it to particular needs.
- Freedom to redistribute copies.
- Freedom to improve the program and release such improvements publicly.

In order to give free software legal copyright protection, the GNU General Public License (GPL) license was created and published in 1989, which gave programmers and users a series of rights and obligations when it comes to the use, modification and distribution of a program and its source code.

Up until that point, the GNU project had matured enough to have produced many programs and utilities that were being used around the globe and secure funds through donations. However, a fundamental piece of the puzzle was still left, the kernel.

Development of an in-house kernel called Hurd had begun some time ago, but its technical complexity was dragging its development significantly.

This stagnation would become less of a problem in 1991, when a man called Linus Torvalds announced publicly that he was developing a kernel based on Minix, a minimal Unix-like operating system used for educational purposes. This kernel, later named Linux, would end up being developed collaboratively by programmers from all over the world in conjunction with Torvalds, reigniting the flame of open collaboration and sharing of code. More and more people started to gain interest in Linux and contribute to its development, which explains how, by 1993, the version 1.0 had been published. Such a fast development enabled companies, as soon as 1994, to start selling machines with the Linux kernel and GNU utilities, proving its potential for wider adoption.

At first, the incentives that Torvalds had for writing Linux were more practical than ideological, since he simply wished for a no cost operating system that he could use and modify. However, soon he discovered the natural consonance with the free software movement and ended up joining forces with the GNU project by licensing the Linux kernel with the GPL license, thus giving birth to the GNU/Linux operating system. In fact, one of the first instances of a GNU/Linux distribution still in use today is Debian, launched in 1994.

However, tensions began to arise among GNU and Linux programmers due to the differing ideas they had on whether utilitarianism should be given a priority over the long-term vision of free software movement. These tensions escalated when, in 1998, a meeting among influential figures of the free and open source community decided to officially create the *open source software* term as an alternative to *free software*. This new term would enable them to convince corporations to adopt a new way of software development without all the heavy ideological connotations that had been attributed to the free software movement. At their root, f*ree software* and *open source software* share the same end goal, but they disagree on the means used for achieving it.

Also in 1998, the Open Source Initiative (27) was created as an organization that provided an umbrella for the open source community. They published the Open Source Definition (28), a document that describes the criteria that software has to comply with to be called open source, which talk about distribution, source code, derivation and licensing.

The 1990s and early 2000s saw the development of many software applications for personal computers and the web such GNOME, KDE, Apache, Samba, PHP, MySQL, OpenOffice, Thunderbird or Firefox, often taking up a significant percentage of the market. This demonstrates how people regained interest in building free and open source software and understand its benefits for everyone, up to the point that, currently, many big companies are heavily invested in funding and developing free and open source software.

3. Container engine architecture

Container engines are the tools that work together to offer containerization functionalities in a consistent and manageable way. Since Docker is one of the most widespread container engines, it will be used as a reference for the analysis of the architecture of a container engine. Knowing the internal structure of a container engine is a prerequisite for understanding which kinds of vulnerabilities may affect any of its components and how to prevent them. Therefore, below we will dive into the structure of a container engine, first from a higher level and then from a lower level perspective.

From a high level view, the Docker engine is structured in a client-server model with several components, such as the Docker client, the Docker daemon, images, containers, volumes and networks. Below, we will explain each of them in further detail.

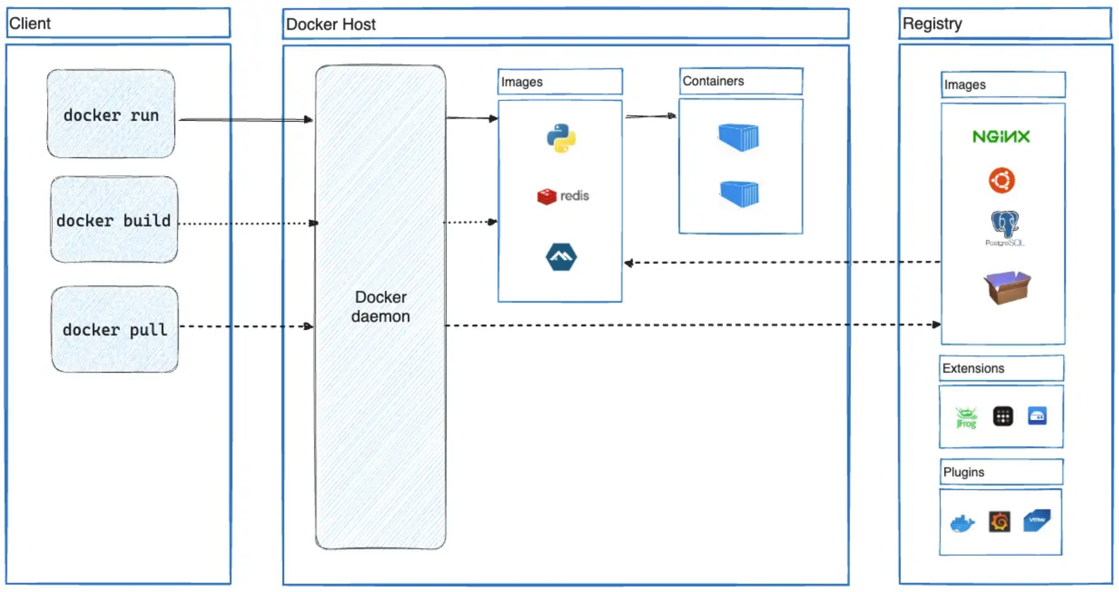

Figure 3: Overview of the Docker engine architecture[3]

The Docker client, *docker,* is a program that acts as an interface between users and the Docker engine, allowing them to generate commands to manipulate the different settings and components of the system. These commands are built using a Representational State Transfer (REST) Application Programming Interface (API) exposed by the Docker daemon and are sent to it via Unix sockets or network interfaces (29).

3 https://docs.docker.com/guides/images/docker-architecture.webp

The Docker daemon, *dockerd*, is a program that sits in a host waiting for commands from the Docker client. These commands are then analyzed and a set of actions are carried out by the proper system components.
Images are immutable templates with instructions that tell Docker how to instantiate a container. They are described on Dockerfiles that specify all the necessary files, binaries, libraries and settings. Images are usually created by combining other images into a new one, similar to a stack made up of many layers. This provides great flexibility, modularity and reusability so that only the necessary features are included. Images can reside locally or be published and downloaded from registries (30).

Containers are executable instances of images materialized as processes that run in an isolated environment with limited resources. The two key Linux kernel features that allow this are *cgroups* and *namespaces*. In short, *cgroups* provide mechanisms for limiting how many system resources a set of tasks can use, whereas *namespaces* are an abstraction that allows to limit which system resources a set of tasks can see (31).

Volumes are a mechanism that allow data to persist independently of the life cycle of containers, while also allowing to back up data more easily, sharing data among multiple containers or storing data in a remote host.

Networks can be created in order to enable Docker containers to communicate among each other inside a host or communicate with other programs on the Internet. A container may be connected to one or more defined networks and may have some ports exposed to the outside. When a network is created, other resources are made available alongside it, such as an Internet Protocol (IP) address, a gateway, a routing table and Domain Name Service (DNS) services.

If we dive deeper into the architecture of the Docker engine, we can see that the Docker daemon delegates some tasks to other components. For instance, while the management of images is mostly performed directly by *dockerd*, the management of the life cycle of containers is carried out by a container runtime, which in the case of Docker is *containerd*.

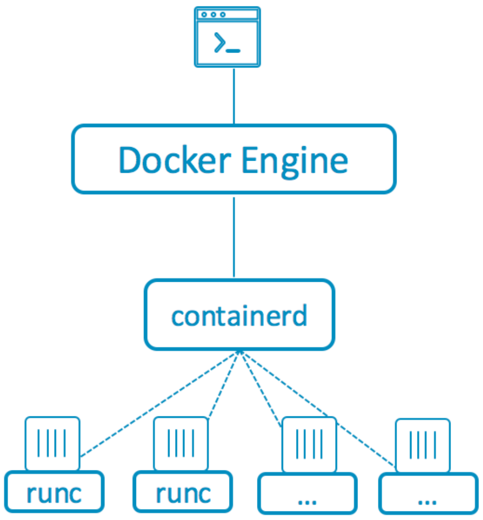

Figure 4: Detailed view of the Docker daemon[4]

Containerd is a high level container runtime that leverages Linux kernel features in order to create an abstraction layer for other tools (like *dockerd*) to manage the life cycle of containers and their resources, such as file systems and networks (32). Its low level counterpart is *runc*, a container runtime based on the Open Container Initiative, which defines the industry standard around container formats and runtimes (33), that is in charge of spawning and running containers.

As an example of the interaction among all the different components, when a user issues a command involving the creation of a container, the command is sent from the Docker client *docker* to the Docker daemon *dockerd*. When *dockerd* receives the command, it validates it, retrieves the image that will be used to create the container and asks *containerd* to perform the necessary actions to create and start the execution of the container. Afterwards, *containerd* allocates the necessary resources for the container and then delegates the actual running of the container to *runc*.

4 https://www.docker.com/wp-content/uploads/2022/12/docker-engine-1-11-runc-1.png

4. Container vulnerabilities

In the previous *State of the art* section, we already introduced a classification of vulnerabilities from a containerization perspective, that is, in which layer of the containerization stack they occur. Now, after having understood the underlying structure of modern containerization technology, it is time to dive deeper into the possible vulnerabilities in each layer.

4.1. Application vulnerabilities

Container application vulnerabilities refer to the vulnerabilities present in the binary and library files of the program that a container runs, which consist of the code and settings that define its functionality.

The amount of vulnerabilities in this category is large, as is the number of applications that use Docker for their deployment. The vulnerabilities present in containerized applications have a more noticeable impact on users, since the integrity and confidentiality of the user’s data is much easily impacted by an exploit.

4.2. Configuration vulnerabilities

Container configuration vulnerabilities occur when the settings related to the permissions, resources and isolation level of containers don't provide the necessary security level for the kind of application running in them.

One area of interest in this regard is the set of Linux kernel capabilities enabled for each container, which define the privileges associated to them. If configured incorrectly, a container could be granted more privileges than required, allowing attackers that get control of the container application to further damage the whole system through privilege escalation (34).

A similar issue regards the execution of the Docker daemon and containers under the root user, which is done primarily in order to leverage various features that are not supported in rootless mode. However, running Docker in a root environment can expose the whole system to exploitation by attackers who manage to break out of the container isolation.

Sidecar containers are a design pattern in which a complementary container runs alongside the main one to support its functionality. Despite their advantage when it comes to modularizing the application logic, they present some dangers in a kind of attacks called sidecar injection, where attackers inject malware inside a sidecar container in order to compromise the whole cluster. One injection method consists in installing *kubectl* inside one of the containers and then running a script to install and run a cryptominer by leveraging the *sh* command. Mitigating these kinds of attacks involves giving containers as few privileges and permissions as possible (35).

There are usually many Docker containers running in parallel on a given physical host and sharing its resources among them. Resource sharing has to be managed carefully by imposing limits on memory space or processing time, since otherwise, attackers could take advantage of loose resource limits to cause a denial of service that could bring the whole system down.

Another important aspect is the connectivity among containers. By default, all containers running in a host are connected to a network, which can be convenient in some circumstances but could also be a security risk if sensitive containers are exposed to untrusted ones.

An often overlooked topic is the correct management of sensitive data. Docker offers mechanisms such as Docker Secrets which provide a method to prevent the exposure of sensitive information in places where attackers could obtain it in order to carry out an attack on the application or the system.

An example of vulnerabilities in this category is CVE-2021-21285[5], a vulnerability in which malformed Docker image manifests crash the Docker daemon and cause uncontrolled resource consumption.

4.3. Image vulnerabilities

Container image vulnerabilities refer to all the vulnerabilities present in the binary and library files and other components of all the layers that make up an image.

In order to run an application inside a container, certain software must be present beforehand. This software could either be other binary or library files that the main application needs. Inevitably, some of these files will contain bugs that, depending on how they affect the application, could render the application or container itself more vulnerable to attacks (36).

In order to minimize the impact that bugs can have on image security, it is important to frequently update images with the latest security patches in order to reduce the time window during which vulnerabilities could be exploited. In addition, creating images as

5 https://www.cve.org/CVERecord?id=CVE-2021-21285

minimal as possible is a good mitigation against exploitations, since by removing unnecessary tools (package managers, network tools, terminal shells, compilers, etc.) the attack surface is significantly reduced.

Since creating new images usually involves using another image as a base, it is essential to check the source of an image. Otherwise, malicious actors could publish their own apparently innocuous images which, instead, contain backdoors. This results in a supply chain attack that could allow them to take control of any container based on that image.

Other issues regarding the integrity of images are related with the place where they are stored. For instance, in Docker, images are stored in public or private registries managed by different organizations. In either case, if the registry is chosen without care, a malicious registry could tamper with the images and compromise the security of any containerized solution that uses those images.

Some examples of vulnerabilities in this category are CVE-2022-42889[6], a vulnerability in the *Text4Shell* library that allows arbitrary code execution; or CVE-2021-44228[7], a vulnerability in the *Log4j 2* library that allows remote code execution for an attacker who can control log messages or message parameters. These vulnerabilities affect several official Docker images, which also impacts derived images and makes updating to a patched version essential to prevent exploits.

4.4. Container engine vulnerabilities

Container engine vulnerabilities involve vulnerabilities found in any of the components of the container engine, with special emphasis on the Docker daemon and the Docker runtime components.

The container engine usually needs to run in an environment with elevated privileges. Accordingly, vulnerabilities affecting it pose a high threat to the security of the whole system since any bug exploitable by a threat actor could allow them to get root privileges on the host.

One way that attackers could access the Docker engine is through the local Unix socket that it listens to. Since this is the main access point to the engine, its exposure to the outside world (for example through a TCP socket) should be considered with extreme care. Otherwise, there is a high risk of an attacker connecting to it and leveraging it to get root privileges on the host (34).

6 https://www.cve.org/CVERecord?id=CVE-2022-42889
7 https://www.cve.org/CVERecord?id=CVE-2021-44228

Another way that attackers could access the Docker engine is by exploiting bugs that allow them to break container isolation. When a threat actor gets access to a container, it can leverage the presence of certain vulnerabilities in the container runtime to escape the isolation layer that prevents access to the main host, which would allow them to compromise the host system with root privileges.

Some examples of vulnerabilities in this category are CVE-2024-41110[8], a vulnerability in the Docker engine that allows an attacker to bypass authorization plugins by means of a specially crafted API request; CVE-2024-21626[9], a vulnerability in *runc* caused by an internal file descriptor leak that allows an attacker to spawn a container with a working directory in the host file system namespace and therefore escape isolation; or CVE-2024-23652[10], a vulnerability in BuildKit that allowed a malicious BuildKit front end or Dockerfile to remove a file outside the container file system.

4.5. Host vulnerabilities

Host vulnerabilities are a kind of vulnerabilities found in the host system on which the container engine runs.

There are many places in a host system where vulnerabilities can reside, such as binary or library files of the different system components, but one of the most critical is the Linux kernel. This is because the abstraction created by a container engine to isolate containers relies on technologies and features present in the Linux kernel, such as *namespaces* and *cgroups*, and therefore, any exploitable Linux kernel vulnerability could end up allowing attackers to break the isolation layer and access the system.

Another fundamental issue related to the security of a host system is network access. In order to interconnect the systems that host all the containers that make up, for example, a solution based on microservices, it is often necessary to employ complex settings to maximize the reliability and performance of the network. Because of this, it is easy to make mistakes and leave holes that attackers can leverage to access the internal network.

Some examples of vulnerabilities in this category are CVE-2022-0492[11], a vulnerability in the Linux kernel *cgroup* functionality that would allow an attacker to escalate privileges and bypass isolation unexpectedly; or CVE-2021-21285[12], a vulnerability in the Linux kernel flags initialization that could allow a regular user to escalate privileges.

8 https://www.cve.org/CVERecord?id=CVE-2024-41110
9 https://www.cve.org/CVERecord?id=CVE-2024-21626
10 https://www.cve.org/CVERecord?id=CVE-2024-23652
11 https://www.cve.org/CVERecord?id=CVE-2022-0492
12 https://www.cve.org/CVERecord?id=CVE-2021-21285

5. Vulnerability scanners

As we saw previously, vulnerability scanners can be classified into 5 categories depending on the features they provide: *application, configuration, image, network* and *host vulnerability scanners*. It is important to keep in mind, however, that this is a theoretical classification, and therefore, actual vulnerability scanning tools often offer a combination of functionalities from different categories.

In order to ensure the security properties of a containerized environment, it is necessary to use a combination of tools capable of dealing with the vulnerabilities present in each layer of the containerization architecture: application, configuration, image, container engine and host system.

In this section, we strive to offer a description of tools that can help deal with vulnerabilities in each layer. Most of them will be specifically tailored to containers, but we also mention more general tools that are worth being taken into consideration.

In the case of application vulnerabilities, we can find tools such as Snyk Code, Snyk Open Source, OSV-Scanner and Nikto.

Snyk (37) is a security platform that enables software developers to secure applications in all phases of the development cycle. One of the tools that it offers is **Snyk Code** (38), a static analyzer that uses a semantic AI-based analysis engine that can analyze various sensitive aspects of the code of an application:

- API usage: It identifies API misuses, null dereferences, type mismatches and use of insecure functions by modeling the use of memory in variables and references.
- Coding issues: It identifies dead code, predefined branches or branches having the same code on each side.
- Control flows: It identifies null dereferences or race conditions by modeling each possible control flow.
- Data flows: It follows the flow of data from the source to the sink to perform taint analysis.
- Hard-coded secrets: It scans the code to detect hard-coded secrets
- Type inference: When dynamically typed languages are used, it is able to determine the initial type of a variable.

- Value ranges: It infers the possible values of variables used to call functions in order to track array errors, division by zero errors and null dereferences.

It is possible to deploy Snyk Code as a full Software as a Service (SaaS) solution, which requires to upload the code to the Internet, deploy it as a self-hosted SaaS solution through a brokered architecture or deploy the engine locally to avoid any code uploads.

Another one is **Snyk Open Source (39)**, a static analyzer specialized in finding and fixing vulnerabilities in the open source libraries used in an application. It is able to scan each open source library and show detailed information of each vulnerability found together with the actions needed to fix it, usually by using pull or merge requests.

A similar tool is **OSV-Scanner** (40), which analyzes a project's list of dependencies and searches its OSV database for vulnerabilities. The OSV database (41) is a distributed and open source database for producing and consuming vulnerability information for open source projects, for vulnerabilities. It works by specifying a root directory from which OSV-Scanner will search lockfiles, Software Bills of Materials (SBOMs) and Git directories to determine the dependencies that need to be checked against the OSV vulnerability database. It can be deployed as a program installed locally or as a Docker container.

For analyzing applications directly facing the Internet, tools like Nikto can be very useful. **Nikto** (42) is a scanner that performs tests against web servers in search of potential risks and security vulnerabilities in items such as server and software misconfigurations, default files and programs, insecure files and programs or outdated servers and programs. It can be installed locally or be run as a Docker container. It works by providing a list of IP addresses and ports to test and, optionally, a set of plugins to be used during the scanning process.

In the case of configuration vulnerabilities, there are useful tools like SecretScanner and Docker Bench for Security.

SecretScanner (43) is a tool that retrieves and searches containers and host file systems and matches their contents against a database of more than a hundred secret types. This helps prevent attackers from accessing sensitive information or credentials that could give access to critical IT infrastructure. SecretScanner is run as a Docker container to which the image, container or host file system to be scanned is provided. The output is written to a JSON file that can be read on the terminal with the help of the *jq* program.

Docker Bench for Security (44) is a script provided by Docker that performs a series of automated tests that check the settings related to the Docker engine and container deployment manifests. It can be run locally as a bash script or through a Docker container. After the tests are finished, the output is saved to a JSON log file with detailed information.

In the case of image vulnerabilities, we have tools such as Grype and Syft (the open source version of Anchore (45)), Snyk Containers, Trivy, Clair and Dagda.

Grype (46) is an image vulnerability scanner that scans file systems, system packages and programming language packages of Docker, OCI and Singularity image formats. Grype can receive input from different programs that generate a Software Bill of Materials (SBOM), which increases its speed. It can be run locally or on a Docker container with options to customize its behavior. The sources of the vulnerability database used by Grype are varied, ranging from Ubuntu Linux Security or Debian Linux CVE Tracker to GitHub Security Advisories, RedHat RHSas, Amazon Linux ALAS or National Vulnerability Databse (NVD).

Syft (47) is a tool that generates SBOMs from Docker, OCI or Singularity container images, file systems and archives. It is specially thought to be combined with Grype, providing details of the packages and dependencies used in software and thus making it easier to manage vulnerabilities, license compliance and software supply chain security.

Snyk Containers (48) is a vulnerability scanner that analyzes operating system and application packages and manifest files of each layer of a container image, looks them up in a vulnerability database and displays the collected information with possible ways to patch the vulnerabilities. The sources of the vulnerability database are both public and private, such as NVD, Debian, Ubuntu or RedHat. Additionally, it provides many integrations with GitHub, Gitlab, Google, Harbor, Microsoft or Amazon image registries.

Trivy (49) is a vulnerability scanner specialized in container and virtual machine images, file systems and Kubernetes and AWS clusters. It scans system and application packages and libraries, configuration files, secrets and software licenses in search of misconfigurations or vulnerabilities. It can be run either locally or on a Docker container, and the results generated can be stored in a text file. In addition, it supports external plugins for enhancing its functionality.

Clair (50) is a vulnerability scanner for Docker and OCI containers. Its architecture is based on the ClairCore library wrapped by a web interface and notification service. The analysis process is made up of the following steps:

- Indexing: When a container image manifest is provided, Clair will separate each layer, scan its contents and generate an intermediate representation called IndexReport.
- Matching: The IndexReport is analyzed and the vulnerabilities described in it are matched with its vulnerability database.
- Notifying: If a new vulnerability is discovered, the notification service will generate an alert if it affects any of the manifests provided to Clair.

The local deployment of Clair can be customized in order to improve its performance, such as dividing the databases used by it or the number of processes it runs in order to distribute the load.

Dagda (51) is a vulnerability scanner that analyzes Docker images and containers to detect vulnerabilities and malware. Its vulnerability database is populated with data from CVEs, Bugtraq IDs (BIDs) Red Hat Security Advisories (RHSAs) and Offensive Security known exploits. Dagda can be run locally or through a Docker container, and when a scan is started, it retrieves information about the operating system and application packages, searches the database for existing vulnerabilities, and stores the results into its database. In addition, Dagda is also able to monitor real-time events of the Docker daemon and running Docker containers to detect anomalous activities.

An important but often overlooked issue in container image security is the protection of the container image registry. According to (52), despite there being several public registries with strong security features, many companies still prefer to host their container images in their own private registries. The problem lies in the fact that many of these registries are not properly secured, leaving sensitive data exposed to the Internet and the door open for malicious image injections. An open source image registry such as **Harbor** (53) can prevent this by scanning and signing the images that are stored in the registry, as well as offering a granular approach to user authentication methods and permissions.

Finally, in the case of container engine and host vulnerabilities, we can use tools such as Falco, cAdvisor, Wazuh and Snort.

Falco (54) is a monitoring and detection tool for Docker containers, hosts and Kubernetes clusters that collects events from various sources, compares them against a set of defined rules and, if an abnormal situation is detected, creates a notification that can be sent to standard output, a file, syslog, a program or an HTTP endpoint. Event sources can go from Linux kernel syscalls to Kubernetes audit logs, AWS cloud events or third party Falco plug-ins. It comes with a predefined set of rules that check the following behavior:

- Privilege escalation using privileged containers
- Namespace changes with tools like *setns*
- Read/write to directories like */etc*, *usrbin*, */usr/sbin*, etc.
- Creation of symlinks
- Changes of ownership and mode
- Unexpected network connections or socket mutations
- Spawned processes using *exeve*
- Executing shell binaries such as *sh*, *bash*, *csh*, *zsh*, etc.
- Executing SSH binaries such as *ssh*, *scp*, *sftp*, etc.
- Mutating Linux *coreutils* executables
- Mutating login binaries
- Mutating *shadowutil* or *passwd* executables suh as *shadowconfig*, *pwck*, *chpasswd*, *getpasswd*, *change*, *useradd*, etc.

cAdvisor (55) is another tool that monitors the resource usage and performance characteristics of running containers. For every container, it collects resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics, aggregates and processes it and then exposes the information through a local web interface. This enables the detection of network and resource usage anomalies that could be a symptom of ongoing exploits. It can be run as a Docker container by providing access to the various system directories it needs to observe.

Wazuh (56) is a security platform that unifies different security functions for endpoints and cloud workloads into a single platform. It can be used for a multitude of use cases, such as monitoring Docker events, detecting unauthorized or hidden processes, detecting operating system vulnerabilities, monitoring file integrity, monitoring system calls, etc. It is made up of four components:

- Agents are deployed on each endpoint, which can be a physical machine or a virtual one, and provide threat detection by monitoring the file system, reading log messages, collecting inventory data, scanning system settings and looking for malware.

- The indexer is a scalable text search and analytics engine that stores data about threat alerts.
- The server processes the data received by the agents and triggers alerts when anomalies are detected.
- The dashboard is a web interface accessible through a browser that works as the interface through which users can analyze and visualize security events and manage the whole platform.

Snort (57) is a tool that can be used as a packet sniffer, a packet logger or a network intrusion prevention system. As the first, it reads the packets from the network and shows them in a continuous stream. As the second, it writes the packet information to a file in disk. As the later, it is capable of performing real-time traffic analysis by carrying out protocol analysis, content search and detection of various attacks such as buffer overflows or stealth port scans. In order to alert of anomalies, Snort is equipped with rules that can be customized to every situation.

6. Vulnerability scanner testing

In this section, we will describe the necessary steps to prepare the environment for the tests and the results obtained after having carried them out.

6.1. Preparation

The system in which we will perform out testing is configured with an Ubuntu 22.04 LTS operating system on an amd64 processor architecture with 8GB of RAM and 512GB of disk memory.

The first step consists in installing the Docker engine, in our case version 27.2.0, and *docker-compose* version 1.29.2.

Afterwards, we download 3 version of 10 of the most popular container images from Docker Hub, which will be used to compare the detection capabilities of each tool. We decided to download 3 versions for each image (published in 2024, 2022 and 2020) so that we can compare how the presence and detection of vulnerabilities changes with older and newer software versions. The list of images together with their specific versions can be found below:

Table 1: Container images and their versions used for testing

IMAGE	YEAR		
	2020	2022	2024
couchbase	6.6.0	7.1.1	7.6.3
httpd	2.4.46	2.4.54	2.4.62
memached	1.6.7	1.6.17	1.6.29
mongo	4.2.9	5.0.13	8.0.0-rc18-noble
mysql	8.0.21	8.0.30	9.0.1
nginx	1.19.2	1.22.1	1.27.1
node	14.8.0	16.17.0	22.7.0
postgres	13	15	16,4
redis	6.0.8	7.0.5	7.4.0
ubuntu	19,1	21,1	24,04

Finally, we need to install and configure the image vulnerability scanners that will be used for our tests: Grype, Trivy, Snyk Containers and Clair. Below we describe the installation and configuration process and offer simple usage instructions.

Grype

Installation

In order to install Grype version 0.80.0, we need to download a script from the project's repository and provide the directory where it will be installed. The command to be executed is as follows:

```
curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh \
    | sh -s -- -b /usr/local/bin
```

Usage

In order to scan an image, it is enough to provide its name as an argument:

```
grype <image>
```

Trivy

Installation

To install Trivy version 0.54.1, we can also perform the same procedure as before with a similar command provided by the project's documentation, which will download the correct binary and put it in the provided directory:

```
curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib \
    /install.sh | sudo sh -s -- -b /usr/local/bin v0.54.1
```

Usage

To scan an image, we need to provide the correct *target* and *subject* to the program. In our case, the *target* will be *image*, and the *subject* will be the name of the image that we want to scan:

```
trivy <target> <subject>
```

Snyk Containers

Installation

In order to install Snyk Containers version 1,1293.0, we need to log in with a GitHub account to its website. After that, we follow the steps indicated by the tutorial, selecting the CLI integration method, downloading and installing the binary in the correct directory following the commands given:

```
curl https://static.snyk.io/cli/latest/snyk-linux -o snyk
chmod +x ./snyk
mv ./snyk /usr/local/bin/
```

and authenticating the machine:

```
snyk auth
```

Usage

In order to scan a container image, we need to provide some arguments together with the image name:

```
snyk container test <repository>:<tag>
```

Clair

Installation

The easiest way to install Clair version 4.7.4, is to clone its GitHub repository in our machine and use *docker-compose* to deploy a local container cluster:

```
git clone git@github.com:quay/clair.git
cd clair
docker-compose up -d
```

This cluster is made of a Quay container, a PostgreSQL container that hosts the vulnerability database, a Traefik container that displays configuration information at address *localhost:8080*, and a Clair container that hosts the core services.

Next, we need to install the *clairctl* tool to interact with the container cluster from the GitHub releases page by downloading the correct binary for our architecture and placing it in the desired directory, such as */usr/local/bin/*.

Finally, we need to place the default *config.yaml* file at the root of the cloned project directory:

```
cp local-dev/clair/config.yaml config.yaml
```

Usage

Now, to use Clair we just need to provide the image name that we want to analyze:

```
clairctl report <image_name>
```

6.2. Results

As we have seen through this project, container security is a vast topic that involves many different aspects such as applications, operating system, networking, etc. For this reason, we decided to focus our testing on a concrete area, container images, which are one of the fundamental blocks of modern container technologies.

The tests that we have carried out involve the use of 4 image vulnerability scanners and 3 different versions of 10 Docker images, as mentioned in the previous section. The procedure consists in executing each tool, analyzing each image and annotating the amount of vulnerabilities discovered by it.

Starting with the images from the year 2024, the results obtained can be visualized in Table 3 and Figure 5. One of the most noticeable features of this diagram is the difference in the amount of detected vulnerabilities among different images. For instance, while the image *node:22.7.0* has around 1200 vulnerabilities, others like *httpd:24.4.62* or *nginx:1.27.1* have at most 150 of them.

The reason for this contrast lies mainly in the fact that more complex images like *node* have more image layers and, therefore, they include more libraries and binaries. The increased amount of files means that there is a greater chance of a vulnerability having been introduced by a programming error, for example. This would explain why images such as *node* have generally more detected vulnerabilities than much simpler images such as *ubuntu*. For this reason, it is a good advice to use a base image that is as small and simple as possible and then add on top of it only the strictly necessary packages needed for the intended purpose of the container.

Table 2: Vulnerabilities detected in image versions from 2024

2024	SCANNER			
IMAGE	Grype	Trivy	Snyk Containers	Clair
couchbase:7.6.3	340	48	119	159
httpd:2.4.62	129	128	67	128
memcached:1.6.29	72	77	38	77
mongo:8.0.0.-rc18-noble	78	22	8	40
mysql:9.0.1	61	8	64	17
nginx:1.27.1	152	157	99	157
node:22.7.0	640	1163	189	1164
postgres:16.4	195	147	56	162
redis:7.4.0	126	77	38	92
ubuntu:24.04	8	6	5	8

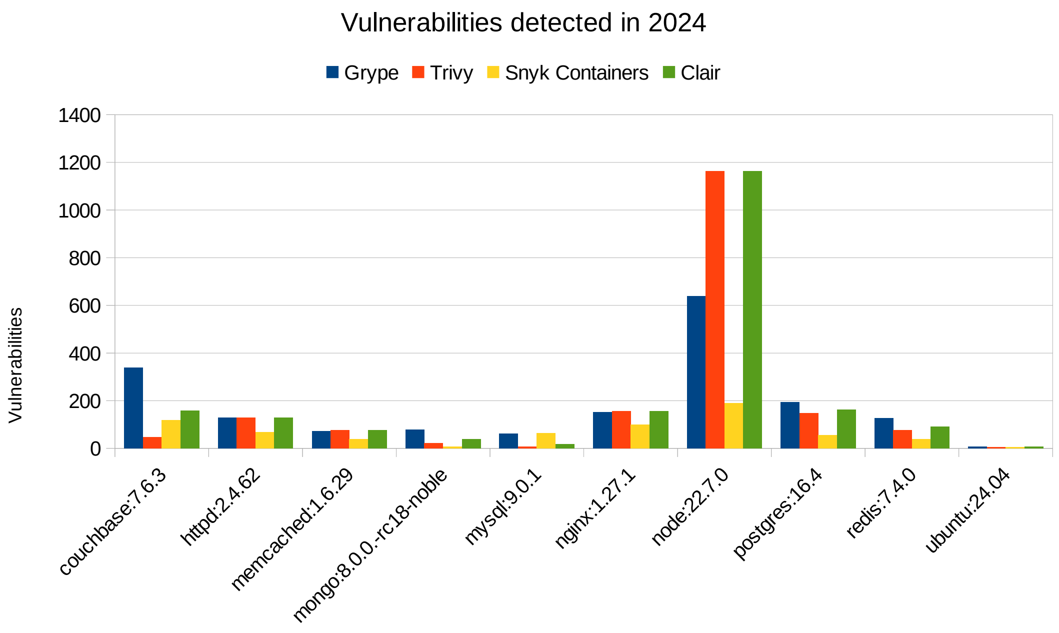

Figure 5: Vulnerabilities detected in image versions from 2024

Moving on, we can observe the results for the image versions from 2022 (Table 3 and Figure 6) and from 2020 (Table 4 and Figure 7). What we can notice in every diagram is that there is a certain variability of the vulnerabilities that each tool is able to detect in each image, which can be clearly seen in for images like *node*, *couchbase* or *postgres*.

The cause for this variation can be attributed to two main factors. One of them is the fact that these vulnerability scanners usually do not use only one database as their source of vulnerabilities but a set of them. Given that different tools use a different combination of source databases, their ability to identify a potential vulnerability as such may vary. The second factor lies in the logic of each vulnerability scanner in change of detecting signs of possible vulnerabilities. Since each tool introduces a different approach to detecting evidences of the presence of vulnerability, the amount of false positives and false negatives may vary.

Table 3: Vulnerabilities detected in image versions from 2022

2022 IMAGE	SCANNER			
	Grype	Trivy	Snyk Containers	Clair
couchbase:7.1.1	2126	129	295	435
httpd:2.4.54	273	254	149	253
memcached:1.6.17	178	181	98	180
mongo:5.0.13	700	253	95	329
mysql:8.0.30	178	174	209	30
nginx:1.22.1	301	305	187	305
node:16.17.0	2914	5005	784	2
postgres:15.0	356	284	124	297
redis:7.0.5	265	182	101	195
ubuntu:21.10	41	8	16	1

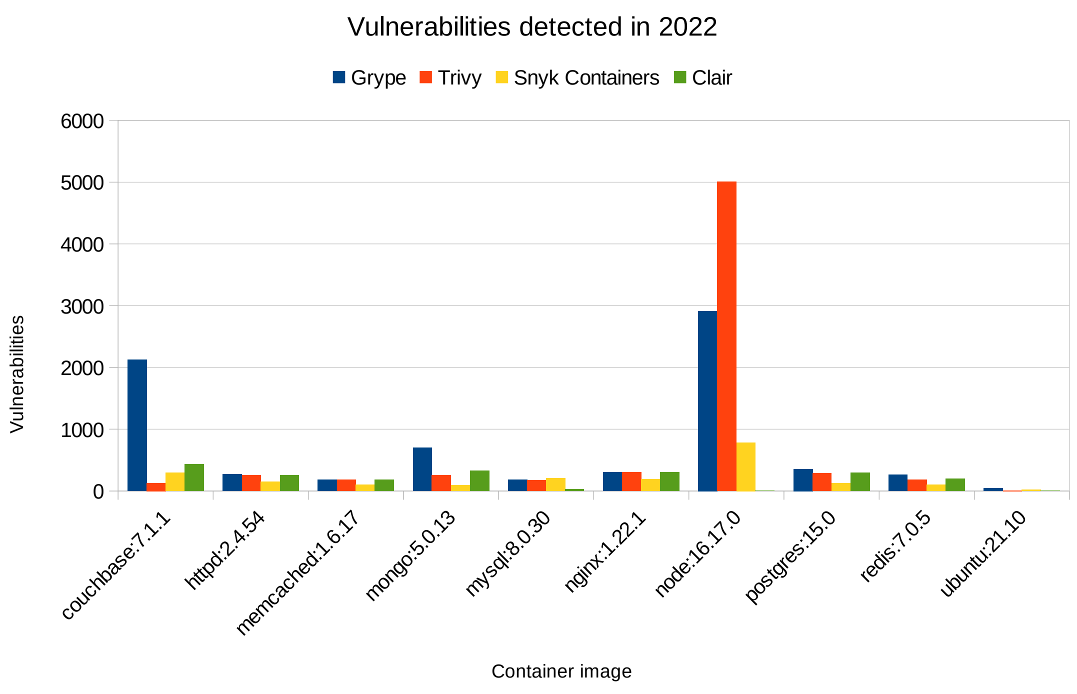

Figure 6: Vulnerabilities detected in image versions from 2022

To illustrate this variability, let’s consider the results for the *ubuntu:21.10* image. While Grype was able to detect 41 vulnerabilities, Trivy could only detect 8 of them. Both agree in the detection of vulnerabilities CVE-22-1304, CVE-2022-34903, CVE-2022-2068 and CVE-2022-2097, but Grype also detected others such as CVE-2022-27943, CVE-2021-46195, CVE-2021-37750 or CVE-2020-16156.

In some cases, the tools are able to indicate, for a vulnerable package, which version solves the vulnerability, so that it can be manually upgraded if necessary. For instance, in the same image as before, the package *e2fsporgs* in version *1.46.3-1ubuntu3* is affected by the vulnerability CVE-2022-1304, and Grype is able to identify that the version *1.46.3-1ubuntu3.1* is no longer vulnerable. Another example is *libssl1.1* version *1.1.1l-1ubuntu1.3*, which has the vulnerability CVE-2022-2068 and CVE-2022-2097. Trivy indicates that while version *1.1.1l-2ubuntu1.5* is no longer vulnerable to the first vulnerability, version *1.1.1l-1ubuntu1.6* also solves the second vulnerability.

An intriguing fact that we have noticed in the tests of the image versions from 2020 is that the vulnerability scanner Clair reports that more than half of the images do not have any vulnerabilities, contrary to the rest of the tools. Our intuition is that Clair might be working on the assumption that, in a production environment, the software in use gets updated at least a couple of times a year, so it is highly unlikely (and we hope so) that an image version from two or four years ago would be used. In such a scenario, Clair can reduce the size of the vulnerability database by populating it with only more recent vulnerabilities, which would explain why it did not detect older ones.

Table 4: Vulnerabilities detected in image versions from 2020

2020	SCANNER			
IMAGE	Grype	Trivy	Snyk Containers	Clair
couchbase:6.6.0	361	117	123	336
httpd:2.4.46	424	404	240	1
memcached:1.6.7	268	280	155	1
mongo:4.2.9	553	339	197	576
mysql:8.0.21	405	422	201	1
nginx:1.19.2	634	667	409	0
node:14.8.0	5015	2851	1028	21
postgres:13.0	523	544	242	1
redis:6.0.8	286	277	154	1
ubuntu:19.10	48	6	3	0

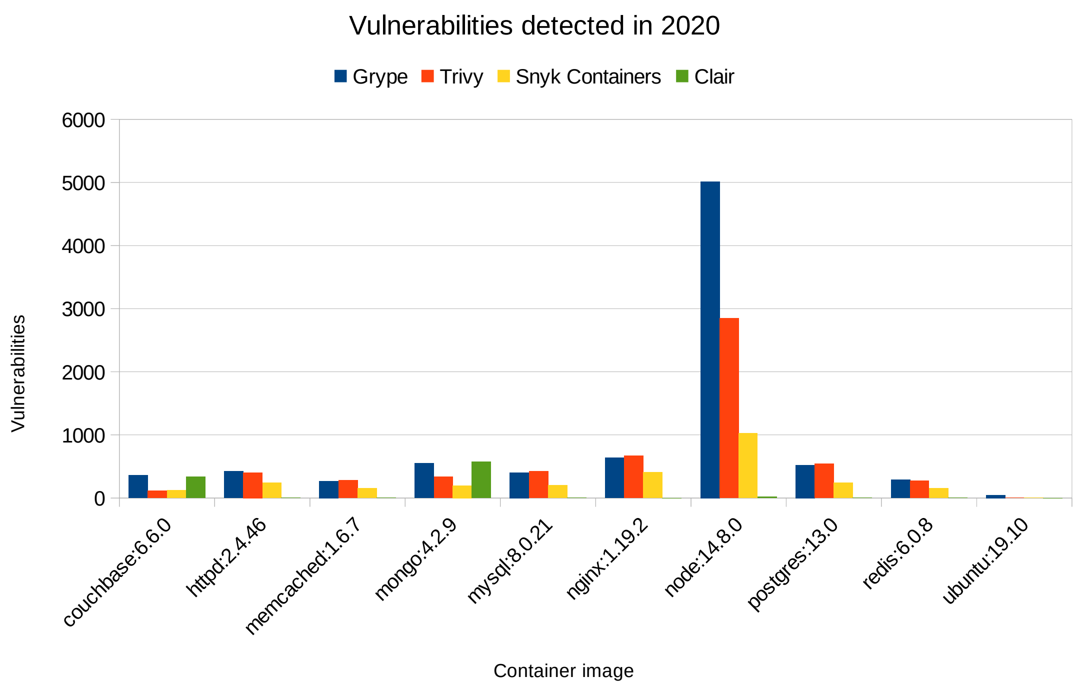

Figure 7: Vulnerabilities detected in image versions from 2020

Since it is difficult to compare the amount of vulnerabilities detected from one year to another in the diagrams above, we have elaborated a new diagram (Figure 8) that shows the sum of all the vulnerabilities detected by each tool grouped by year. As one would expect, the general observed trend is that the older a piece of software is, the higher the chance is of finding new vulnerabilities in it. This demonstrates that it is a valid advice to insist in updating software regularly and as soon as possible in order to prevent the exploitation of old and new vulnerabilities, which are highly valued by attackers.

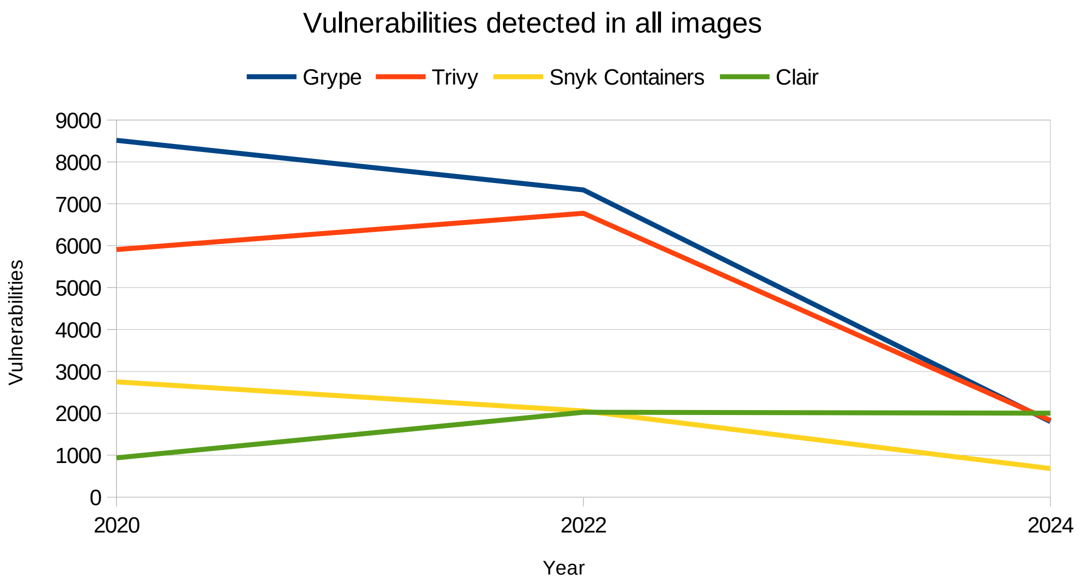

Figure 8: Vulnerabilities detected in all images by year

It may be convenient to also make some comments on the time performance of each tool. The scanning process of the image vulnerability scanners used in our tests can be divided into two parts.

The first part consists in checking whether the vulnerability database is updated. If it is not, the vulnerability database update is retrieved from the Internet, a process that may vary in time. For instance, for tools like Grype or Trivy it may take up as much as 1 or 2 minutes (depending on the Internet connection), whereas for Clair it can take up as much as 5 minutes. Snyk is a special case here, since it performs the analysis on its own servers online. Because of this, the command line tool does not store a local copy of the database and the update process on the servers happens transparently to the user.

The second part consists in actually analyzing and querying the database, a process that mostly depends on the size of the image. For example, an image like *httpd* is much smaller compared to *node*, so the time employed in analyzing it is going to be lower.

In order to illustrate these differences, let’s consider the time it takes the four tools we have used before to analyze two images of varying size, *httpd:2.4.62* and *node:22.7.0*. In the first diagram (Figure 9) we can observe the execution time of each tool when a database update (if applicable) needs to be performed and when it does not for the image *httpd:2.4.62*.

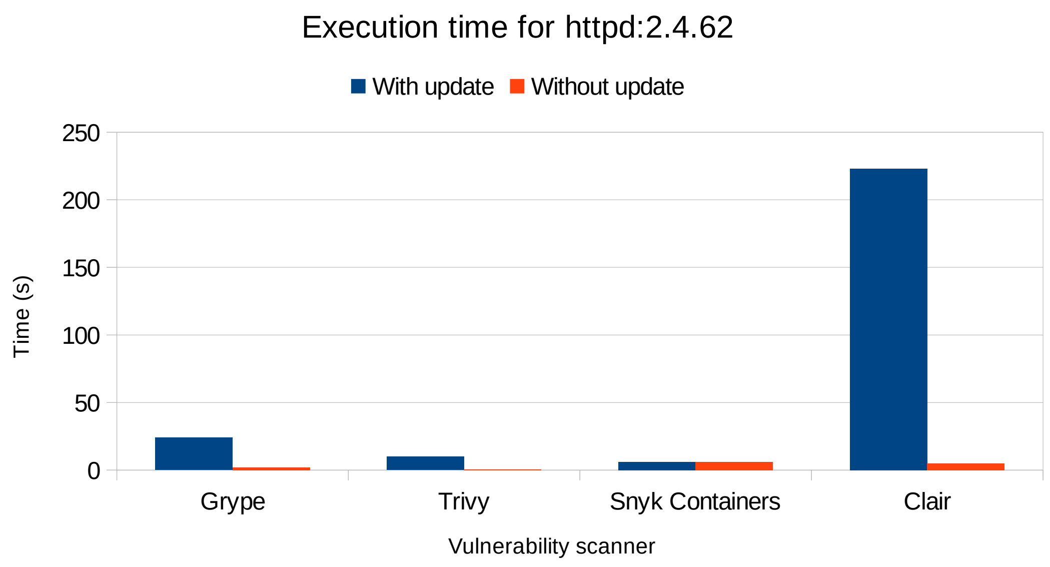

Figure 9: Execution time for httpd:2.4.62

As we can appreciate, even though Grype and Trivy have some noticeable differences in execution time (24s vs 2s and 10s vs 0,5s), Clair has a large difference mainly due to its long update time (223s vs 5s).

In the case of the execution time for the image *node:22.7.0* (Figure 10), we observe a similar proportion between the results where an update in needed and results where an update is not, although the time it takes for Trivy and Snyk to analyze the *node* image is around 3 times longer and for Grype and Clair it is 6 and 7 times longer, respectively.

Finally, as we assumed, the execution time of Snyk does not vary, since it does not store any local database.

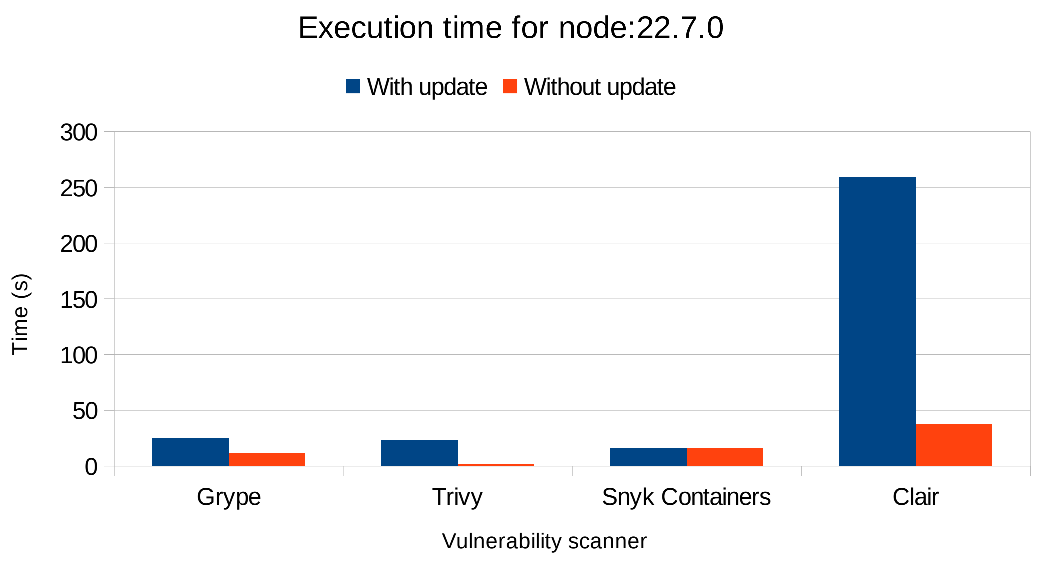

Figure 10: Execution time for node:22.7.0

7. Conclusions

Container technologies are one of the key enablers of the cloud computing revolution thanks to the easier management and the more efficient resource utilization they offer, allowing the deployment of new services and platforms at a much larger scale.

In this project, we have seen how the technological context around containers has evolved, from the origin of virtualization technologies, the increasing importance and complexity of computer security and the philosophical and industrial revolution of free and open source software, creating a paradigm of better collaboration and interoperability between technologies.

We have also explored the intricacies of how a modern containerization engine operates, which parts it is made of, and how they interact with each other to offer a seamless and practical solution for creating and managing containers.

In addition, we have categorized the different types of container vulnerabilities depending on where they appear in the containerization software stack, described their properties and discovered the ways in which they can be tackled, thanks to open source tools that offer various capabilities tailored to different types of vulnerabilities.

In order to understand how these tools actually perform in real life scenarios, we have carried out some practical tests with popular open source vulnerability scanners and container images. The results have shown a noticeable variability in the detection capabilities of each tool depending on a combination of factors.

The best recommendations for an improved container security state lies in using containers as up to date as possible, reducing the size of containers to the bare minimum and using a combination of vulnerability analysis tools in all stages of software development, from code analyzers to runtime monitoring tools, all in the effort of reducing the amount of vulnerabilities that end up being exploited.

The future work regarding our project would include other experiments that could allow us to identify the number of true and false positives and negatives or the accuracy of the tools. Such experiments would involve crafting a custom container image based on an image without vulnerabilities, introducing each vulnerability manually in order to be certain about the exact amount of them, and comparing the detections of each tool.

8. References

1. RedHat [Internet]. 2024 [cited 2024 Aug 8]. Understanding virtualization. Available from: https://www.redhat.com/en/topics/virtualization

2. Strachey CS. Time sharing in large, fast computers. In: Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959. UNESCO (Paris); 1959. p. 336–41.

3. Randal A. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. ACM Comput Surv. 2020 Feb 6;53(1):5:1-5:31.

4. Fedoseenko V. A brief history of virtualization, or why do we divide something at all [Internet]. 2019 [cited 2024 Jul 1]. Available from: https://www.ispsystem.com/news/brief-history-of-virtualization

5. Kerrisk M. User namespaces progress [Internet]. 2012 [cited 2024 Aug 24]. Available from: https://lwn.net/Articles/528078/

6. Corbet J. Notes from a container [Internet]. 2007 [cited 2024 Aug 24]. Available from: https://lwn.net/Articles/256389/

7. IBM [Internet]. 2023 [cited 2024 Jul 1]. What Is Virtualization? Available from: https://www.ibm.com/topics/virtualization

8. CISA [Internet]. 2021 [cited 2024 Aug 8]. What is Cybersecurity? Available from: https://www.cisa.gov/news-events/news/what-cybersecurity

9. Codecademy [Internet]. 2024 [cited 2024 Jul 1]. The Evolution of Cybersecurity. Available from: https://www.codecademy.com/article/evolution-of-cybersecurity

10. CWE - Common Weakness Enumeration [Internet]. 2024 [cited 2024 Jul 1]. Available from: https://cwe.mitre.org/index.html

11. CVE - Common Vulnerabilities and Exposures [Internet]. 2024 [cited 2024 Jul 1]. Available from: https://www.cve.org/

12. Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput. 2004 Jan;1(1):11–33.

13. CISA [Internet]. 2024 [cited 2024 Aug 19]. KEV - Known Exploited Vulnerabilities Catalog. Available from: https://www.cisa.gov/known-exploited-vulnerabilities-catalog

14. Ben-Ari D. Panorays. 2023 [cited 2024 Jul 1]. What You Need to Know About Known Exploited Vulnerabilities. Available from: https://panorays.com/blog/known-exploited-vulnerabilities/

15. CVSS - Common Vulnerability Scoring System [Internet]. 2024 [cited 2024 Jul 1]. Available from: https://www.first.org/cvss/v4.0/specification-document

16. CWE: Research Concepts [Internet]. 2024 [cited 2024 Jul 31]. Available from: https://cwe.mitre.org/data/definitions/1000.html

17. 2023 CWE Top 25 Most Dangerous Software Weaknesses [Internet]. 2023 [cited 2024 Jul 14]. Available from: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

18. Constantin L. CSO online. 2020 [cited 2024 Jul 1]. What are vulnerability scanners and how do they work? Available from: https://www.csoonline.com/article/569221/what-are-vulnerability-scanners-and-how-do-they-work.html

19. Snyk [Internet]. 2024 [cited 2024 Aug 20]. Everything You Need to Know About Container Scanning. Available from: https://snyk.io/learn/container-security/container-scanning/

20. Ehrman N. Wiz. 2023 [cited 2024 Aug 20]. Container Security Scanning Explained. Available from: https://www.wiz.io/academy/container-security-scanning

21. Andew D. Intruder. 2024 [cited 2024 Jul 1]. What is Vulnerability Scanning? Available from: https://www.intruder.io/guides/the-ultimate-guide-to-vulnerability-scanning

22. Tozzi C. For Fun and Profit: A History of the Free and Open Source Software Revolution [Internet]. The MIT Press; 2017 [cited 2024 Aug 9]. Available from: https://direct.mit.edu/books/book/3528/For-Fun-and-ProfitA-History-of-the-Free-and-Open

23. Free Software Foundation [Internet]. 2024 [cited 2024 Aug 18]. Free Software Foundation. Available from: https://www.fsf.org/index.html

24. Stallman R. GNU. 2021 [cited 2024 Aug 18]. The GNU Manifesto. Available from: https://www.gnu.org/gnu/manifesto.html

25. Stallman R. GNU. 2024 [cited 2024 Aug 18]. What is Free Software. Available from: https://www.gnu.org/philosophy/free-sw.html

26. Stallman R. Free software, free society: selected essays. 1st. ed. Gay J, editor. Boston, Mass: Free Software Foundation; 2002. 220 p.

27. Open Source Initiative [Internet]. 2024 [cited 2024 Aug 18]. Open Source Initiative. Available from: https://opensource.org/

28. Open Source Initiative [Internet]. 2006 [cited 2024 Aug 13]. The Open Source Definition. Available from: https://opensource.org/osd

29. Docker Documentation [Internet]. 2024 [cited 2024 Jul 1]. Docker Documentation. Available from: https://docs.docker.com/

30. Türkal F. How does Docker actually work? The Hard Way: A Technical Deep Diving [Internet]. Medium. 2024 [cited 2024 Jul 1]. Available from: https://medium.com/@furkan.turkal/how-does-docker-actually-work-the-hard-way-a-technical-deep-diving-c5b8ea2f0422

31. Linux manual pages [Internet]. 2024 [cited 2024 Jul 3]. Linux man pages online. Available from: https://man7.org/linux/man-pages/index.html

32. Ostrowski S. Docker. 2024 [cited 2024 Jul 3]. Containerd vs. Docker: Understanding Their Relationship and How They Work Together. Available from: https://www.docker.com/blog/containerd-vs-docker/

33. Open Container Initiative [Internet]. 2024 [cited 2024 Jul 3]. Open Container Initiative. Available from: https://opencontainers.org/

34. OWASP [Internet]. 2024 [cited 2024 Jul 15]. Docker Security - OWASP Cheat Sheet Series. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

35. Logan M. Trendmicro. 2024 [cited 2024 Aug 3]. Mitigating the Threat of Sidecar Container Injection. Available from: https://www.trendmicro.com/vinfo/fr/security/news/security-technology/mitigating-the-threat-of-sidecar-container-injection

36. RedHat [Internet]. 2020 [cited 2024 Jul 15]. Container Image Security: Beyond Vulnerability Scanning. Available from: https://www.redhat.com/en/blog/container-image-security-beyond-vulnerability-scanning

37. Snyk Documentation [Internet]. 2024 [cited 2024 Aug 21]. Snyk. Available from: https://docs.snyk.io/

38. Snyk Documentation [Internet]. 2024 [cited 2024 Aug 28]. Snyk Code Documentation. Available from: https://docs.snyk.io/scan-using-snyk/snyk-code

39. Snyk Documentation [Internet]. 2024 [cited 2024 Aug 28]. Snyk Open Source Documentation. Available from: https://docs.snyk.io/scan-using-snyk/snyk-open-source

40. OSV-Scanner [Internet]. Google; 2024 [cited 2024 Aug 21]. Available from: https://github.com/google/osv-scanner

41. OSV databse [Internet]. 2024 [cited 2024 Aug 26]. Available from: https://osv.dev/

42. Nikto [Internet]. Nikto; 2023 [cited 2024 Aug 28]. Available from: https://github.com/sullo/nikto/wiki/Home

43. SecretScanner [Internet]. Deepfence; 2024 [cited 2024 Aug 21]. Available from: https://github.com/deepfence/SecretScanner

44. Docker Bench for Security [Internet]. Docker; 2024 [cited 2024 Aug 21]. Available from: https://github.com/docker/docker-bench-security

45. Anchore [Internet]. 2024 [cited 2024 Aug 26]. Anchore open source tools. Available from: https://anchore.com/opensource/

46. Grype [Internet]. Anchore, Inc.; 2024 [cited 2024 Aug 21]. Available from: https://github.com/anchore/grype

47. Syft [Internet]. Anchore, Inc.; 2024 [cited 2024 Aug 21]. Available from: https://github.com/anchore/syft

48. Snyk Container Documentation [Internet]. 2024 [cited 2024 Aug 29]. Available from: https://docs.snyk.io/scan-using-snyk/snyk-container

49. Trivy [Internet]. Aqua Security; 2024 [cited 2024 Aug 21]. Available from: https://github.com/aquasecurity/trivy

50. Clair [Internet]. QUAY; 2024 [cited 2024 Aug 21]. Available from: https://github.com/quay/clair

51. Grande E. Dagda [Internet]. 2024 [cited 2024 Aug 21]. Available from: https://github.com/eliasgranderubio/dagda

52. Dreher C. dreher.in. 2024 [cited 2024 Aug 30]. Unprotected container registries. Available from: http://dreher.in/blog/unprotected-container-registries

53. Harbor [Internet]. Harbor; 2024 [cited 2024 Aug 30]. Available from: https://github.com/goharbor/harbor

54. Falco [Internet]. Falco; 2024 [cited 2024 Aug 21]. Available from: https://github.com/falcosecurity/falco

55. cAdvisor [Internet]. Google; 2024 [cited 2024 Aug 21]. Available from: https://github.com/google/cadvisor

56. Wazuh [Internet]. Wazuh; 2024 [cited 2024 Aug 21]. Available from: https://github.com/wazuh/wazuh

57. Snort [Internet]. Snort 3.0 Team; 2024 [cited 2024 Aug 21]. Available from: https://github.com/snort3/snort3

9. Annexes

Annex 1: Sustainable Development Goals

Degree of relationship of the project with the Sustainable Development Goals (ODGs):

Table 5: Relationship between the project and the SDGs

Sustainable Development Goal	High	Medium	Low	Not related
1. No Poverty				x
2. Zero Hunger				x
3. Good Health and Well-Being		x		
4. Quality Education		x		
5. Gender Equality				x
6. Clean Water and Sanitation				x
7. Affordable and Clean Energy				x
8. Decent Work and Economic Growth		x		
9. Industry, Innovation and Infrastructure		x		
10. Reduced Inequalities				x
11. Sustainable Cities and Communities		x		
12. Responsible Consumption and Production				x
13. Climate Action		x		
14. Life Below Water				x
15. Life on Land				x
16, Peace, Justice and Strong Institutions				x
17. Partnerships				x

The 2030 Agenda for Sustainable Development was adopted by all United Nations member states in 2015, providing a road map for peace and prosperity. It is based on the 17 Sustainable Development Goals, which define strategies for ending poverty, improving health and education, reducing inequality, preserving the environment and encourage economic growth.

Public administrations (such as schools, universities, governments or hospitals) are one of the most targeted entities in cyberattacks, party due to the huge impact on the society that an interruption of operation can cause. Our project can have a positive impact on *Good Health and Well-Being*, *Quality Education* and *Sustainable Cities and Communities* by promoting a better understanding of the security risks of the systems deployed and how to deal with them so that they are more resilient against attacks.

Every day, an increasing number of industries are adopting a microservice architecture for the deployment of their systems, which means that usually they will be using a containerization solution like Docker. Understanding the causes and the impact of attacks against the infrastructure and how to detect and mitigate them is essential to ensure the continued development of new technologies and the creation of new opportunities, which is reflected in *Industry, Innovation and Infrastructure* and *Decent Work and Economic Growth*.

Finally, containers allow a better server utilization because they simplify how they are shared among different users and applications. This allows for a reduction in the high energy usage attributed to data centers and cloud architectures, thus having a positive impact on *Climate Action*.